Cobblestone Graduate Software Engineer Role Assignment Code:

```
import math
import random
import time

# Function to simulate a continuous data stream
def data_stream():
    t = 0
    while True:
        # Introduces a seasonal pattern using a sine wave function
        seasonal = math.sin(2 * math.pi * (t % 100) / 100)

        # Adds random noise with a mean of 0 and a standard deviation of 0.1
        noise = random.gauss(0, 0.1)

        # Initially, no anomaly
        anomaly = 0

        # With a 5% probability, inject an anomaly between 1 and 3
        if random.random() < 0.05:
            anomaly = random.uniform(1, 3)

        # The final data point combines seasonal, noise, and anomaly
        data_point = seasonal + noise + anomaly

        # Yield the data point, simulating a continuous stream
        yield data_point

        # Increment time step
        t += 1

# Class for detecting anomalies using Exponential Weighted Moving Average (EWMA)
class AnomalyDetector:
    def __init__(self, alpha=0.3):
        """
        Initializes the AnomalyDetector.
```

```
        Args:
        - alpha (float): Smoothing factor for EWMA, controls the weight
given to the most recent data.
        """
        self.alpha = alpha  # Smoothing factor
        self.ewma = None    # Initial EWMA is set to None
        self.std_dev = 0    # Initial standard deviation
        self.n = 0          # Counter for the number of data points
processed

    def update(self, data_point):
        """
        Updates the EWMA and checks if the data point is an anomaly.
        Args:
        - data_point (float): The current value from the data stream.

        Returns:
        - bool: True if the data point is an anomaly, False otherwise.
        """
        if self.ewma is None:
            # Initialize EWMA with the first data point
            self.ewma = data_point
            self.std_dev = 0
        else:
            self.n += 1
            prev_ewma = self.ewma

            # Update the EWMA using the exponential smoothing formula
            self.ewma = self.alpha * data_point + (1 - self.alpha) *
self.ewma

            # Update the standard deviation using an incremental formula
            self.std_dev = math.sqrt(((self.n - 1) * self.std_dev**2 +
(data_point - prev_ewma)**2) / self.n)

        # Threshold is set at 3 times the standard deviation
        threshold = 3 * self.std_dev

        # If the difference between data point and EWMA exceeds the
threshold, it's an anomaly
```

```python
        is_anomaly = abs(data_point - self.ewma) > threshold
        return is_anomaly

# Function to visualize the data stream in a text-based manner
def visualize_stream(detector):
    """
    Visualizes the data stream and detected anomalies.
    Args:
    - detector (AnomalyDetector): The anomaly detector instance that
checks for anomalies.
    """
    data_stream_gen = data_stream()  # Start data stream generation

    print("Data Stream (Anomalies marked with '*'):")
    for i, data_point in enumerate(data_stream_gen):
        # Update the anomaly detector with each new data point
        is_anomaly = detector.update(data_point)

        # Print data point, flagging anomalies with an asterisk (*)
        if is_anomaly:
            print(f"{i:3}: {data_point:.5f}  * Anomaly")
        else:
            print(f"{i:3}: {data_point:.5f}")

        # Simulate real-time streaming with a slight delay
        time.sleep(0.05)

        # Stop after 500 data points to limit the demonstration
        if i > 500:
            break

# Create an instance of AnomalyDetector with alpha=0.3
detector = AnomalyDetector(alpha=0.3)

# Visualize the data stream and detect anomalies in real-time
visualize_stream(detector)
```

Algorithm Explanation:

The **Exponential Weighted Moving Average (EWMA)** algorithm is used for anomaly detection in the streaming data. It updates a smoothed average (EWMA) based on a smoothing factor (alpha), giving more weight to recent data points. The algorithm compares the difference between the current data point and the EWMA to the standard deviation. If the difference exceeds a set threshold (3 times the standard deviation), the point is flagged as an anomaly.

Why EWMA is Effective:

- **Adapts to Concept Drift**: EWMA adapts to changes in data patterns by adjusting the average based on recent trends.
- **Handles Noise**: The standard deviation helps differentiate anomalies from random noise.
- **Efficient for Streaming**: EWMA is computationally light and works well for continuous data streams.

Key Points:

1. **Data Stream Simulation**: Generates data with seasonal patterns, noise, and random anomalies.
2. **Anomaly Detection**: Detects anomalies in real-time using EWMA.
3. **Text-Based Visualization**: A simple real-time display of data points, highlighting anomalies.

Code Optimization:

Sliding Window Approach:

The sliding window approach processes a fixed-size subset of data from a continuous stream. As new data arrives, the oldest data is discarded, and the newest data is added, maintaining a "window" of the most recent data points. This technique is commonly used for real-time processing and analysis of data streams where handling the entire dataset isn't feasible due to memory or performance constraints.

Code:

```
import math

import random

import time

from collections import deque

import matplotlib.pyplot as plt
```

```python
# Function to simulate a continuous data stream
def data_stream():
    t = 0
    while True:
        # Seasonal pattern using a sine wave function
        seasonal = math.sin(2 * math.pi * (t % 100) / 100)

        # Adds random noise with a mean of 0 and a standard deviation of 0.1
        noise = random.gauss(0, 0.1)

        # Injects anomaly with a 20% probability (for testing purposes)
        anomaly = 0
        if random.random() < 0.2:  # Increased from 5% to 20%
            anomaly = random.uniform(1, 3)

        # Combine seasonal, noise, and anomaly into the final data point
        data_point = seasonal + noise + anomaly

        # Yield the data point, simulating a continuous stream
        yield data_point
        t += 1
```

```python
# Class for detecting anomalies using EWMA and sliding window

class AnomalyDetector:

    def __init__(self, alpha=0.3, window_size=50):

        """

        Initializes the AnomalyDetector.

        Args:

        - alpha (float): Smoothing factor for EWMA, controls the weight 
given to recent data.

        - window_size (int): Size of the sliding window for real-time 
anomaly detection.

        """

        self.alpha = alpha

        self.ewma = None

        self.std_dev = 0

        self.window_size = window_size

        self.window = deque(maxlen=window_size)  # Buffer to store sliding 
window of data points



    def update(self, data_point):

        """

        Updates the sliding window, EWMA, and checks if the data point is 
an anomaly.

        Args:

        - data_point (float): The current value from the data stream.
```

```
        Returns:

        - bool: True if the data point is an anomaly, False otherwise.

        """

        # Add the new data point to the sliding window

        self.window.append(data_point)



        # Initialize EWMA with the first data point in the window

        if len(self.window) == 1:

            self.ewma = data_point

            self.std_dev = 0

            return False  # Cannot detect anomalies on the first point



        # Update EWMA using the sliding window

        prev_ewma = self.ewma

        self.ewma = self.alpha * data_point + (1 - self.alpha) * self.ewma



        # Calculate standard deviation for the window

        mean_diff = sum([(x - prev_ewma)**2 for x in self.window]) /
len(self.window)

        self.std_dev = math.sqrt(mean_diff)



        # Threshold is set at 2 times the standard deviation (increased
sensitivity)

        threshold = 2 * self.std_dev
```

```
        # If the difference between the data point and EWMA exceeds the threshold, it's an anomaly
        is_anomaly = abs(data_point - self.ewma) > threshold

        return is_anomaly


# Function to visualize the sliding window and detected anomalies using matplotlib
def visualize_stream(detector):
    """
    Visualizes the data stream and detected anomalies with a sliding window using matplotlib.
    Args:
    - detector (AnomalyDetector): The anomaly detector instance that checks for anomalies.
    """
    data_stream_gen = data_stream()  # Start data stream generation
    window_size = detector.window_size

    # Initialize the plot
    plt.ion()  # Interactive mode on
    fig, ax = plt.subplots()
    data_x = []
    data_y = []
```

```
    anomaly_x = []  # Store x-values of anomalies

    anomaly_y = []  # Store y-values of anomalies


    for i, data_point in enumerate(data_stream_gen):

        # Update the anomaly detector with each new data point

        is_anomaly = detector.update(data_point)


        # Get the current window of data

        current_window = list(detector.window)

        data_x.append(i)

        data_y.append(data_point)


        # Clear the plot

        ax.clear()


        # Plot the data points

        ax.plot(data_x, data_y, label="Data Stream", color='gray',
alpha=0.6)


        # Highlight the sliding window data points in blue

        if len(data_x) > window_size:

            ax.plot(data_x[-window_size:], data_y[-window_size:],
label="Sliding Window", color='blue')
```

```python
        # Highlight the current data point if it's an anomaly
        if is_anomaly:
            anomaly_x.append(i)
            anomaly_y.append(data_point)
            print(f"Anomaly detected at time step {i}: {data_point}")

        # Plot all detected anomalies
        if anomaly_x:
            ax.plot(anomaly_x, anomaly_y, 'ro', label="Anomalies")

        # Set plot labels and title
        ax.set_xlabel('Time Step')
        ax.set_ylabel('Data Value')
        ax.set_title('Real-Time Data Stream with Sliding Window and Anomaly Detection')

        # Add legend
        ax.legend()

        # Pause for a short interval to simulate real-time data streaming
        plt.pause(0.1)

        # Stop after 500 data points to limit the demo
        if i > 500:
```

```
            break


    plt.ioff()  # Interactive mode off

    plt.show()


# Create an instance of AnomalyDetector with alpha=0.3 and sliding window size of 50

detector = AnomalyDetector(alpha=0.3, window_size=50)


# Visualize the data stream and anomalies in real-time with sliding window

visualize_stream(detector)
```

Pros of Sliding Window Approach:

1. **Memory Efficiency**: Only a limited number of data points are stored, making it suitable for large or infinite data streams.
2. **Real-Time Processing**: Focuses on the most recent data, making it effective for detecting patterns or anomalies in real time.
3. **Fast Computation**: Processing is restricted to the window size, ensuring low computational complexity.
4. **Handles Concept Drift**: Adapts to changes in data trends by constantly updating with the latest information.

Cons of Sliding Window Approach:

1. **Loss of Historical Data**: Older data outside the window is discarded, potentially missing long-term patterns or trends.
2. **Window Size Sensitivity**: The choice of window size is critical. A small window may miss important information, while a large window may introduce unnecessary noise or lag.
3. **Not Suitable for All Scenarios**: Applications that require the full history or long-term trends may not benefit from the sliding window approach.

Error Handling and Validation:

- The code uses basic error handling by ensuring that `EWMA` is initialized correctly when the first data point is processed.
- Data validation could be added to ensure that incoming data points are valid numbers (e.g., not `None` or non-numeric values).

No External Libraries Used:

Since the project limits external libraries, the code avoids using `matplotlib` and instead provides a text-based output, which works for the project's purpose of detecting anomalies in a streaming environment.